Secure Interrupts on Low-End Microcontrollers

Ruan de Clercq*‡, Frank Piessens†‡, Dries Schellekens*‡, and Ingrid Verbauwhede*‡
*KU Leuven, Department of Electrical Engineering (ESAT), COSIC, Belgium
†KU Leuven, Department of Computer Science, DistriNet, Belgium
‡iMinds, Belgium

***Abstract*—Embedded devices are increasingly becoming interconnected, sometimes over the public Internet. This poses a major security concern, as these devices handle sensitive information (e.g., banking credentials, personal data) or they are critical for the safety of human lives (e.g., smoke detector, airbag system). Security protocols need to be used in combination with a trusted computing base to ensure that attackers cannot alter the state of the software running on these devices to leak secrets. In this work we focus on the problem of secure interrupt handling, which has not been covered in related work. Our architecture for secure interrupts build on the idea of using simple memory isolation techniques to ensure leakage free processing of secret information on a microcontroller. Three methods of securely handling interrupts are proposed, each exploring a different trade-off between hardware and software complexity, and interrupt latency. Prototype implementations based on an openMSP430 softcore demonstrate the practical feasibility of our architecture.**

I. Introduction

Embedded devices permeate our everyday lives. These devices are increasingly becoming interconnected, either through local ad-hoc networks or with the public Internet. This improved connectivity brings new opportunities: the functionality can be updated remotely by the manufacturer, greatly improving time-to-market, as well as allowing software from third party providers to be installed on demand. However, this also raises security concerns, especially if the device processes sensitive information or performs safety critical tasks.

Today's embedded systems lack the security technologies that are present in modern, general-purpose platforms, such as secure boot, virtualization, measured launch, etc. Even basic processor features, such as virtual memory and privilege levels (i.e., separation between user and kernel mode), are missing in low-end microcontroller units (MCUs). The isolation of different tasks has to be enforced by an operating system without any hardware support. The complexity of these operating systems is growing, especially if they need to include a network stack.

In recent work, researchers have proposed hardware changes to improve the trustworthiness of low-end MCUs by adding a lightweight root of trust. Kumar et al. [1] designed a system that provides a number of protection domains within the address space and introduced the idea of using a safe stack to guard against stack corruption by misbehaving tasks. Strackx et al. [2] proposed a simple program counter based memory access control system to isolate software modules. The SMART architecture [3] supports dynamic remote attestation with a software routine stored in immutable ROM. Noorman et al. [4] designed a security architecture called Sancus, that supports strong process isolation and hardware based remote attestation. Schulz et al. [5] designed a security architecture called TrustLite, that provides hardware-enforced isolation of software modules with support for secure exception handling, and communication between protected modules.

ARM TrustZone [6] is a technology that is available on a number of commercially available ARM processors. Each processor has two virtual processors, with different privileges and a strictly controlled communication interface. Intel Software Guard Extensions (Intel SGX) [7] provide ISA extensions that provide multiple protected domains, called enclaves, in which software can operate, free from external observation or modification of the code and data. However, these solutions are only available on high-end processors.

The majority of the related works, which are applicable to low-end MCUs, require their security functionality to execute uninterruptedly. For SMART this is a strict requirement, otherwise an attacker can move malware around during attestation to avoid detection [8].

Embedded devices regularly operate in a real-time environment and need to remain responsive to external stimuli (e.g., a medical implant, or a safety system in a car), even while performing time-consuming security-critical tasks (e.g., generation of cryptographic key or digital signature). Preemptive schedulers can provide real-time scheduling support for MUCs. It works by temporarily interrupting a task, without requiring its cooperation, and resuming it at a later stage. Secure scheduling is useful to enable the on-schedule execution of critical tasks. The scheduler is isolated from other software by keeping it inside a protected domain, and it makes use of interrupts to perform scheduling. Secure I/O processing can be used to perform leakage-free processing of received I/O data in the protected domain. When I/O data arrives, interrupts are used to invoke a routine that is located in the protected domain. The routine can then process the I/O data without leaking it's contents to the other software running on the processor.

In this work, we provide a mechanism for handling secure interrupts that is compatible with the Sancus architecture. The proposed architecture allows *interrupt service routine* (ISRs) to be located in either the secure or the non-secure domain of the microcontroller. This makes it possible to use the processor for real-time processing, secure scheduling, and secure IO, as tasks running in any security domain can be interrupted. We present a generic solution and compare the results of three implementation options.

The paper is structured as follows. First, we present the general architecture of our security-enhanced MCU, which is based on design principles of the related work. Next, we describe a general scheme that can be used to allow secure interrupts. Subsequently, we discuss three implementations with different design trade-offs. Finally, we discuss our implementation results, and present the conclusion.

II. Architecture

This section discusses the security architecture of the system. We first describe the attacker possibilities, followed by a description of the security enhancements for domain isolation and security domain switching.

A. Attacker model

For the attacker model we assume that the adversary is capable of obtaining full control of the state of the software and data. This has the following implications. First, the attacker is capable of modifying any writable code, e.g., a buffer overflow attack. Second, the attacker can read, and write to any memory region that is not explicitly protected by the MCU. Third, the attacker may have compromised the underlying layer of software, e.g., the OS.

We also assume that the attacker is not capable of performing any hardware-based attacks, including placing probes on the memory bus, performing a hard reset of the system, and inducing hardware faults.

B. Domain isolation

The system is partitioned into two different domains, like in [3], [6]: (1) the *non-secure domain* where regular activities occur, and (2) the *secure domain* where all processing of sensitive data occurs. If an operating system is present on the embedded device, it will typically reside in the non-secure domain; this system for instance contains a network stack or a real-time scheduler. Both the program memory and the data memory are partitioned into their respective secure and non-secure parts. For the sake of simplicity, we only consider a single secure domain, but our scheme can easily be extended to multiple secure domains [1], [2], [4].

The MCU makes a distinction between the two domains depending on the program counter (PC). When the PC is in the address range of the secure program memory, the system is considered to be inside the secure domain. When the PC is in the address range of the normal program memory, then the system is considered to be in the normal domain.

We assume a low-end microcontroller without a memory management unit (MMU) and hence no support for virtual memory. Instead a basic *memory protection unit* (MPU) is inserted between the MCU and the memory. This unit enforces (1) *program counter based access control* [2] on both the data memory and the program memory, and (2) guards the entry into the secure domain by allowing only a single point of entry.

The program counter based access control feature ensures that only the normal program memory and data memory is accessible while the system is in the normal domain. However, when the system is inside the secure domain, both the secure program memory and secure data memory also becomes accessible to the processor.

The single point of entry into the secure domain, from hereon referred to as the *single entry point*, is a mechanism that ensures that the secure domain can only be entered at a single address which is located in the secure program memory. Once the program counter has entered the secure program memory at this address, it is allowed to transition to any other address inside either the secure, or non-secure program memory. However, once the program counter points to an address that lies outside of the secure program memory, the secure domain can only be entered again via the single entry point. This feature ensures that secure code cannot be (ab)used to extract secure data. In return oriented programming [9] an attacker selectively combines chunks of secure program memory, which could lead to unintended information leaks. The hardware enforcement of the single entry point guards against such type of attacks.

The enforced access control is shown in Table I. The secure data memory is inaccessible when the MCU is in the non-secure domain. Furthermore, the non-secure domain only has read access to the secure program memory, except for its first address. This memory address has execute permission, and acts as the only entry point to the secure domain.

In order to expose multiple functions from the secure domain to the non-secure domain, a *jump table* is used, as also proposed in [1], [4]. The identifier of a specific function is stored in a register before jumping to the single entry point. The code at the single entry point then jumps to the correct function based on the identifier passed inside the register.

TABLE I
Access rights enforced by the memory protection unit.

	Program Memory		**Data Memory**	
	non-secure	**secure**	**non-secure**	**secure**
Secure domain	`rwx`	`rwx`	`rw-`	`rw-`
Non-secure domain	`rwx`	`r-x`*	`rw-`	`---`

*Execute access only available on the single point of entry

C. Context switching between domains

We define a *domain switch* as a transition from one security domain to another. Context switches within a domain (e.g., multithreading, user/kernel mode switching) are not considered in this work.

Two type of context switches can be distinguished. The first type are instructions that alter the program counter, including the `call` instruction, a return from a call with the `ret` instruction, a return from interrupt (`reti`), or with the `jmp` instruction. The second type are hardware events such as interrupts, processor exceptions or a reset.

The memory protection unit enforces the isolated memory regions of the two security domains. However, these domains still share the same set of registers. Therefore, special care is needed such that information does not leak through registers.

Consequently, the general-purpose registers need to be cleared before a domain switch to the non-secure domain.

A call stack is typically used to pass parameters, store return addresses, and for local data storage. A microcontroller maintains a *stack pointer* (SP) that points to the top of the stack. In our solution, each domain has its own stack, which is located in its data memory address space, and a dedicated SP register. The MCU switches between the two registers depending on the security domain. We chose this option for the sake of simplifying domain switches, and providing better performance. An alternative method, which is used in [4], is to perform stack pointer switching in software by storing pointers to the top of each stack in fixed data memory addresses; this solution requires only a single hardware register.

III. Secure Interrupts

This section discusses the scheme for handling secure interrupts. We first describe a typical interrupt mechanism of low-end MCUs and then present a modified scheme for supporting interrupts with multiple security domains.

A. Standard interrupt mechanism

An interrupt is a signal generated by hardware or software to indicate to the processor an event that needs immediate attention (e.g., timer, peripheral device, etc.).

Each interrupt can have its own unique interrupt service routine (ISR). The addresses of the ISRs are stored in the *interrupt vector table* (IVT), which is located at a specific program memory address.

When an interrupt occurs, the following steps are generally performed: (1) the currently executing instruction is completed, (2) the PC that points to the next instruction (which we call the *resume point*), is stored on the stack, (3) the status register (SR) (which contains the status flag bits, e.g., zero, carry, and overflow) is pushed on the stack, (4) the interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction, (5) the SR is cleared and further interrupts are disabled, and (6) the address stored in the IVT is loaded into the PC, causing a jump to the ISR. When the ISR is finished, it resumes the interrupted task with the `reti` instruction. This instruction restores the SR and PC from the stack to continue execution at the point where it was interrupted.

Some processor architectures supported *nested interrupt*. In this case, interrupts can be re-enabled inside an ISR, causing any interrupt that occurs inside this ISR to interrupt the routine, regardless of its priority.

B. Domain isolation support

We have extended the microcontroller with the notion of isolated protection domains. As mentioned above, it is crucial that no information leakage occurs during a domain switch from the secure to non-secure domain. With instruction based domain switching, the content of registers is cleared in software just before the transition occurs. However, this strategy cannot easily be applied with an interrupt based domain switch.

There are three main design challenges. First, a hardware interrupt can occur at any point during execution. This implies that there are four possible scenarios: (1) a non-secure task is interrupted by a non-secure ISR, (2) a secure task by a secure ISR, (3) a non-secure task by a secure ISR, and (4) a secure task by a non-secure ISR. The first two are trivial to handle, as no domain switch is required; however, the latter two require a domain switch.

Second, it should be possible for the software to choose whether to resume the interrupted task, or to execute another task. This allows for the scheduling of secure/non-secure tasks, as will be explained in section III-E.

Finally, the scheme must still comply with the hardware restricted entry into the secure domain. The ISR is unaware whether it is interrupting a secure or non-secure task. Normally it will resume execution with the `reti` instruction. However, this instruction cannot be used to directly perform a domain switch into the secure domain, as it would invalidate the single entry policy. If this restriction was not in place, then it could lead to an attacker circumventing the single entry policy by pushing a secure domain address onto the stack, followed by issuing a `reti`.

C. Interrupting non-secure task with secure ISR

In this scenario, an ISR that resides in secure program memory, is invoked from within the non-secure domain. Here we propose to handle this scenario with the scheme shown in Fig. 1. The secure domain is entered at the single entry point. We propose to solve the problem of invoking the secure ISR from the non-secure domain by adding an entry to the single entry jump table (section II-B) for each secure ISR. Each entry is responsible for invoking a different ISR. The general-purpose registers should be cleared before a domain switch from the secure domain to the non-secure domain. However, the transition into the secure domain does not require the general-purpose registers to be cleared, because they do not contain any secrets.

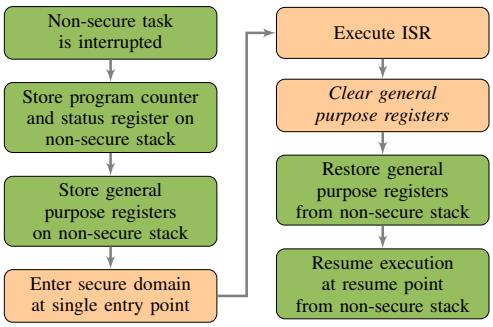

Fig. 1. A general scheme for invoking secure ISRs from the non-secure domain. The green blocks reside in the non-secure domain, whereas orange blocks reside in the secure domain.

D. Interrupting secure task with non-secure ISR

In this scenario, an ISR that resides in non-secure program memory, is invoked from the secure domain. Here we propose to handle this scenario with the scheme shown in Fig. 2. A domain switch from the secure domain to the non-secure domain is required. Therefore, the general-purpose registers need to be cleared before switching to the non-secure domain. We propose to solve the problem of resuming the interrupted secure task from the non-secure domain, by adding an entry into the single entry jump table (section II-B) to resume execution at the resume point.

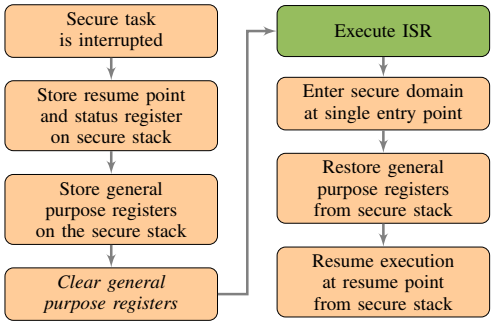

Fig. 2. A general scheme for invoking non-secure ISRs from the secure domain. The green blocks reside in the non-secure domain, whereas orange blocks reside in the secure domain.

E. Scheduling

Microcontrollers often have real-time operating requirements where scheduling of tasks are essential. When using a preemptive scheduler, interrupts are generated with a hardware timer, to transfer control back to the scheduler. The scheduler then selects the next task to execute, or resume, based on a ranking system. The mechanism that allows for interrupting a secure task with a non-secure ISR also enables preemptive scheduling.

A secure scheduler, as described in [10], enables the on-schedule execution of critical tasks that are running on a partially compromised system. This type of scheduler prevents components under the attacker's control from changing the execution times of other applications. The scheduler is kept isolated from the rest of the software by placing it inside the secure domain. When an application is preempted or an exception occurs, control is transferred to the scheduler which resumes execution of the pending applications. Therefore, the mechanism that allows for interrupting a non-secure task with a secure ISR also enables secure scheduling.

IV. Implementation

This section presents the implementations that were made in order to demonstrate the feasibility of our proposed scheme. First, we describe the architecture of the processor that we extended with security features. Afterwards, we present three prototype implementations with different design goals, and different design trade-offs in terms of cycles, area, and code size.

A. MSP430

Our implementation is based on the low-cost, low-power TI MSP430. It is a 16-bit MCU with a Von Neumann architecture, and a single 16-bit address space for program and data memory. It has no external memory bus, and the amount of on-chip memory is limited to 16 kB RAM and 256 kB flash memory. It has eleven general-purpose registers (R4-R15), with R0-R3 serving as a program counter, stack pointer, status register, and constant generator.

Most interrupts on the MSP430 architecture are maskable, and can therefore only cause an interrupt when they are enabled, and if the general interrupt enable (GIE) bit is set inside the status register.

A multiplexer is used to select between the normal SP and the secure SP, depending primarily on the value of the program counter.

One of the design problems we faced, was that ISRs have a return control flow that depends on the domain that the ISR was invoked from. We decided that each ISR should use the same return mechanism, regardless of the domain it is invoked from. We solve this by invoking all ISRs that require a domain switch in a special manner, which we call an *emulated interrupt*. Instructions are used to emulate what the hardware does when an interrupt occurs, by pushing the SR and the address of a *return trampoline* onto the stack, followed by a jump to the ISR. The invoked ISR executes, and returns with a `reti`. Since the address of the return trampoline is still on the stack, the `reti` will invoke the return trampoline.

B. Software-based implementation

The goal of the software-based implementation is to use the minimum amount of hardware features. As such, we opted to make use of a different IVT for each security domain. Since the ISRs have control flows that depend on the current security domain, the idea is that each IVT will serve as the starting point for each of these control flows.

An IVT is normally populated with the addresses of ISRs, each associated with a different interrupt. As we now have an IVT for each security domain, we populate it with (1) the addresses of ISRs that exist in the same security domain as the IVT, and (2) the addresses of software routines that will initiate the control flow to invoke ISRs located in the other security domain.

The second IVT, which we refer to as the *secure IVT*, is stored at a fixed address in secure program memory. A hardware feature selects between IVTs, depending on the current security domain.

Upon entering the secure domain at the single point of entry, the value stored in R15 is used to determine which function in the jump table to execute.

1) Interrupting non-secure task with secure ISR: The architecture is shown in Fig. 3. After storing the general-purpose registers, a `call` is made to the entry in the single entry jump table that corresponds to the current ISR. The secure ISR is invoked with an emulated interrupt. This is followed by clearing the general-purpose registers, and returning (`ret`)

from the `call`, which causes a transition back to the normal domain. The registers are then restored from the secure stack, and the resume point is used to jump to the point where execution was interrupted.

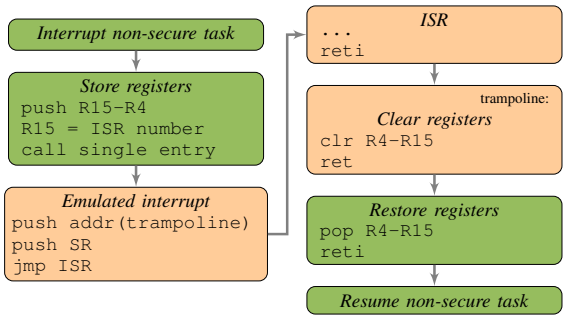

Fig. 3. Software-based flowchart for invoking a secure ISR from the non-secure domain. The green blocks reside in the non-secure domain, whereas orange blocks reside in the secure domain.

2) Interrupting secure task with non-secure ISR: The architecture is shown in Fig. 4. After storing, and clearing the general-purpose registers, the non-secure ISR is invoked by means of an emulated interrupt. This is followed by storing a value of zero in R15, and jumping to the single entry point. The value of zero in R15 correspond to a jump table entry that is responsible for restoring all registers from the secure stack, and using the resume point to jump to the point where execution was interrupted.

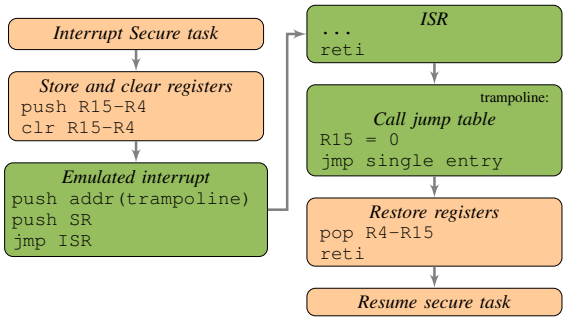

Fig. 4. Software-based flowchart for invoking a non-secure ISR from the secure domain. The green blocks reside in the non-secure domain, whereas orange blocks reside in the secure domain.

C. Hardware-based implementation

The goal for the hardware-based implementation is to minimize interrupt latency by adding more functionality to hardware. As a further goal we try to minimize the amount of additional hardware. The architecture of this implementation is very similar to the software-based implementation (section IV-B), with the exception of the following: (1) a single IVT is used, and (2) the clearing, storing, and restoring of general-purpose registers is now done in hardware.

A single IVT is used to store the addresses of the ISRs that can be located in either domain. The extended interrupt logic, shown in Fig. 5, takes care of any additional processing that needs to be done if a domain switch is required to invoke an ISR.

Fig. 1 and 2 show that both schemes require: (1) saving the general-purpose registers to the stack, and (2) restoring the general-purpose registers from the stack after domain switching back to the interrupted task's domain. We propose to solve (1) by further extending the interrupt logic to save the general-purpose registers on the stack before performing the domain switch to the other domain. We further propose to solve (2) by introducing a new instruction that restores all registers from the current stack.

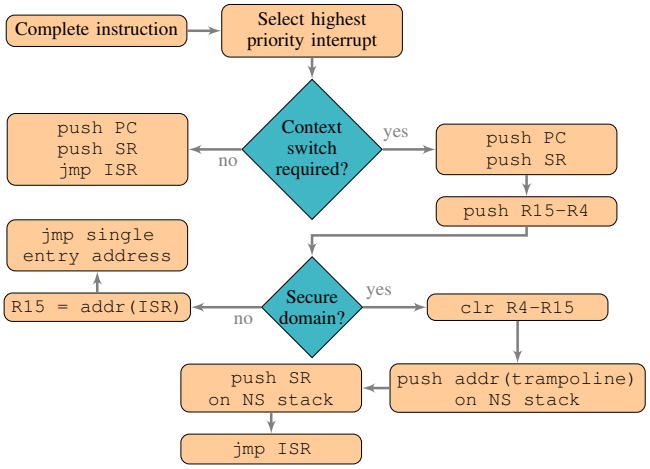

Fig. 5. The modified hardware-based interrupt logic.

D. Hidden registers optimization

We also propose a performance optimization to improve interrupt latency. For this optimization, the amount of visible registers inside the ISR is restricted to n. This improves interrupt latency because less registers need to be cleared, saved and restored from the stack, but without the hardware cost of using shadow registers. To ensure that the remaining $11 - n$ general-purpose registers do not leak any information, all read and write operations on the remaining registers will be blocked. The compiler/programmer needs to ensure that only the visible n registers are used in the ISR, as the remaining registers are unusable.

This optimization can be done on either the hardware-based architecture (section IV-C), or the software-based architecture (section IV-B), and will require the following features to be activated when inside a domain switched ISR: (1) the saving and restoring of the general-purpose registers on the stack is limited to only n registers, and (2) the register file is modified to disable read and write access on the remaining registers.

V. Results

We extended the openMSP430 [11] softcore to create our prototype implementations. This softcore is fully compatible with the TI MSP430 MCU, and executes code generated by any MSP430 toolchain in a near cycle-accurate way. The openMSP430 softcore was configured to use the following

settings: a 10 MHz clock, 4 kB of data memory, 8 kB of program memory, a hardware multiplier, and a single timer. We used the Digilent Atlys, Spartan-6 LX45 based FPGA development board to test our results.

Table II shows the amount of 6-input LUTs, and registers for the unmodified openMSP430, and the three different designs. The synthesis optimization goal was set to "area".

TABLE II
A SUMMARY OF THE HARDWARE COSTS FOR THE DIFFERENT DESIGNS.

Design	LUTs	Registers
Unmodified	2231	1185
Software-based	2241	1187
Hardware-based	2417	1219
HW-based Hidden registers	2420	1220

Table III compares the number of cycles required to perform an interrupt-based context switch. An interrupt that does not require a domain switch needs 6 cycles to pass control to the ISR, whereas 5 cycles are required to resume an interrupted task with the `reti` instruction. Pushing a register onto the stack requires 3 cycles, whereas popping a value from the stack requires 2 cycles. For the cycle times it is assumed that jump table logic requires only 4 cycles.

TABLE III
CONTEXT SWITCHING CYCLE TIMES FOR AN INTERRUPTED TASK. THE NUMBER OF HIDDEN REGISTERS IS INDICATED WITH n .

Context switch	SW	HW
Interrupt S to NS	71	9+n
Return NS to S	43	14+n
Interrupt NS to S	54	13+n
Return S to NS	52	10+n
Interrupt *	6	6
Return *	5	5

* No domain switch occurs here.

The results show that the software oriented technique has the slowest context switching time, and further requires the least amount of hardware modifications. Our hardware-based design has a moderately good context switching time with a slightly bigger hardware cost. The hidden register method proved to have the fastest context switching time, at the cost of a small amount of additional hardware, and a reduced number of available registers in the ISR.

VI. CONCLUSION

In this paper, we proposed a scheme that allows interrupts to be securely processed on a processor that uses program counter based isolation techniques to protect sensitive data. We designed three implementations, each with different design trade-offs, and made the prototypes by extending the openMSP430 softcore. In all three cases, both hardware, as well as software techniques are required to make the prototypes.

Our result show that secure interrupts are feasible on low-end microcontrollers as the amount of extra hardware that is required is minimal, and the cycle overhead from domain switches is acceptable.

ACKNOWLEDGMENT

This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007). In addition, this work is supported in part by the Flemish Government, FWO G.00130.13N, FWO G.0876.14N, the Hercules Foundation AKUL/11/19, and by Intel.

REFERENCES

[1] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. B. Srivastava, "A System For Coarse Grained Memory Protection In Tiny Embedded Processors," in *Design Automation Conference*, 2007, pp. 218–223.

[2] R. Strackx, F. Piessens, and B. Preneel, "Efficient Isolation of Trusted Subsystems in Embedded Systems," in *Security and Privacy in Communication Networks*, 2010, pp. 344–361.

[3] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust," in *Network and Distributed System Security Symposium*, 2012.

[4] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base," in *USENIX Security Symposium*, 2013, pp. 479–494.

[5] P. K. S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *EuroSys*. ACM, 2014.

[6] *ARM Security Technology - Building a Secure System using TrustZone Technology*, ARM Limited, 2009.

[7] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," *HASP*, vol. 13, p. 10, 2013.

[8] A. Francillon, Q. Nguyen, K. B. Rasmussen, G. Tsudik, J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "A minimalist approach to remote attestation," in *Design, Automation and Test in Europe*, 2014.

[9] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the Difficulty of Software-based Attestation of Embedded Devices," in *ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 400–409.

[10] R. J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun, "Enabling trusted scheduling in embedded systems," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 61–70.

[11] "OpenCores project web site," http://www.opencores.org/, 2014, accessed: 2014-02-20.